Massachusetts Institute of Technology

Artificial Intelligence Laboratory

AI Memo No. 551 August 1980

AN OUTLOOK ON TRUTH MAINTENANCE

by

David A. McAllester*

Abstract:

Truth maintenance systems have been used in several recent problem solving systems to record justifications for deduced assertions, to track down the assumptions which underlie contradictions when they arise, and to incrementally modify assertional data structures when assumptions are retracted. A TMS algorithm is described here that is substantially different from previous systems. This algorithm performs deduction in traditional propositional logic in such a way that the premise set from which deduction is being done can be easily manipulated. A novel approach is also taken to the role of a TMS in larger deductive systems. In this approach the TMS performs all propositional deduction in a uniform manner while the larger system is responsible for controlling the instantiation of universally quantified formulae and axiom schemas.

Keywords: Theorem Proving, Automated Deduction, Truth Maintenance, Backtracking, Dependencies, Assumptions, Likelihood, Demonic Invocation, Hierarchy

This report describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643 and in part by National Science Foundation Grant MCS77-04828.

* IBM Fellow

Acknowledgments

There are many people who supported and encouraged this work. Gerald Sussman and Jon Doyle originally guided me into the area of truth maintenance and belief revision. Chuck Rich, Jerry Roylance, Howie Shrobe, and Ken Forbus provided many stimulating discussions.

CONTENTS

1. Introduction **1**

2. The TMS **2**

2.1 Propositional Constraint Propagation 2
2.2 Justifications 4
2.3 Retraction 6
2.4 Backtracking and Refutation 7

3. Premise Control **10**

3.1 Assumptions 10
3.2 Likelihood Classes 11
3.3 Likelihood Assertions 13

4. Instantiation Control **14**

4.1 For-All Assertions 14
4.2 Taxonomic Hierarchies: An Example 15
4.3 Areas for Further Research 17
4.3.1 Demonic Invocation 17
4.3.2 Special Purpose Subsystems 18

5. Relation to Other Work **19**

Appendix I: The TMS Code **21**

Appendix II: Utility Procedures **32**

Function Index **40**

References **42**

1. INTRODUCTION

Recently there has been some interest in dependency structures which can be used in reasoning and problem solving systems. Such dependency structures can be used to make reasoning systems more incremental during retraction and backtracking and better able to explain their results [Fikes 75][Stallman & Sussman 77][Shrobe 77]. This has given rise to special purpose systems for handling logical dependencies, and even the notion of non-monotonic logics, i.e. logics in which larger premise sets can have fewer valid consequences [Doyle 77][Reiter 79]. I have borrowed the term truth maintenance system (TMS) from Jon Doyle to describe any system with the following four characteristics:

a) It performs some form of propositional deduction from a set of premises.

b) It maintains justifications and explains the results of its deductions.

c) It incrementally updates its beliefs when premises are added or removed.

d) It does dependency directed backtracking; i.e. when a contradiction arises it uses the recorded justifications to track down the premises which underlie that contradiction.

Two major points about truth maintenance are made here. First a TMS can be based on deduction in traditional propositional logic. Second, a TMS can be used as an active *deductive* component of general deductive systems.

The TMS algorithm described here is based on "propositional constraint propagation" which was originally described, in essence, by Davis and Putnam [Davis & Putnam 60]. This technique is related to the algebraic constraint propagation of Sussman [de Kleer & Sussman 78] and the graph labeling algorithms of Waltz [Waltz 72]. Section two describes the details of this technique, and the way in which the basic TMS functions listed above can be integrated into it.

The ability to incrementally manipulate the premise set allows a great deal of flexibility. Assumptions can be made which can later be retracted if they are found to support contradictions. Some basic techniques for controlling assumptions, and for controlling the premise set in general, are described in chapter three.

The TMS can be used to perform all propositional deduction in a general deduction framework. The primary aspect of general deduction which cannot be performed by a TMS is the instantiation of quantified formula and axiom schema. The position is taken here that those problems which are of a purely propositional nature can be solved to such a degree that the only difficult issues remaining in automated deduction involve the control of instantiation.

2. THE TMS

The TMS described here operates on an assertional data base. Traditionally an assertional data base has been either a simple set of assertions or a collection of "contexts" each of which could be thought of as a set of assertions [Hewitt 72] [McDermott 74]. The TMS algorithm developed here operates on a data base in which assertions are assigned to one of the three states, "true", "false", and "unknown". It is possible to map this data base to a simple set of assertions by first taking the set of assertions which are "true" and then adding the negations of the assertions which are "false". Thus the data base can be simultaneously viewed as a traditional set of assertions, and as an assignment of the states "true", "false", and "unknown" to non-traditional assertions.

These different outlooks on the assertional data base can lead to some confusion in terminology. For example suppose one wants to assert that Fred is not a fish. One can speak of "adding the premise" **(not (fish Fred))**. But this act of "adding a premise" may actually involve putting the assertion **(fish Fred)** in a "false" state. The remainder of this paper relies on context for the proper interpretation of such terminology.

There are four primary functions performed by this TMS. First it performs propositional deduction via a deduction technique which is termed here "propositional constraint propagation". Second it generates justifications for each deduced truth value. Next it is capable of incrementally updating the data base when premises are removed. Finally it is capable of dependency directed backtracking. The technique used in backtracking is easily extended to a refutation mechanism which adds to the deductive power of the TMS.

2.1. Propositional Constraint Propagation

In general a constraint propagation system has a set of "cells" which can take on values, and a set of "constraints" which constrain those values. Whenever a new value follows from the previously determined values and a *single* constraint, this value is deduced. In what will be termed "simple" constraint propagation these are the only deductions which are made. Constraint propagation terminates when there are no further deductions which can be made from single constraints, and the set of constraints is said to be "relaxed". If the number of values which can be determined by a single constraint is bounded then this process can take no longer than linear time in the number of constraints.

In propositional constraint propagation the assertions in the data base are viewed as cells or "TMS nodes" which can take on one of the values "true" or "false". All logical relations (constraints) in the TMS take the form of disjunctive clauses such as **(v (p . false) (q . true))**. This constraint says that is impossible for both P to be true and Q to be false. Therefore whenever P is true the constraint could be used to deduce that Q must be true. Likewise whenever Q is false the constraint could be used to deduce that P must be false. As in the case of general constraint propagation, the TMS will deduce any truth value which follows from the truth values already present and a single constraint (clause). This process is iterated until the constraint set is relaxed. It is possible to add premises and constraints incrementally and the TMS has no difficulty performing the additional deduction needed to relax the constraint set.

Table I. Axioms for Propositional Logic

```
or      (v ((or p q) . false) (p . true) (q . true))
        (v ((or p q) . true) (p . false))
        (v ((or p q) . true) (q . false))

and     (v ((and p q) . true) (p . false) (q . false))
        (v ((and p q) . false) (p . true))
        (v ((and p q) . false) (q . true))

->      (v ((-> p q) . false) (p . false) (q . true))
        (v ((-> p q) . true) (p . true))
        (v ((-> p q) . true) (q . false))

not     (v ((not p) . true) (p .true))
        (v ((not p) . false) (p . false))
```

It is important to note that *not all* deductions which follow logically from a set of constraints are deduced by this algorithm. For example, **p** follows from the two constraints, **(v (p . true) (q . true))** and **(v (p . true) (q . false))**. However the system can make no deductions from these constraints. Some of the implications of this observation, and some ways of dealing with such situations are discussed in later sections of this chapter.

The TMS performs propositional deduction from a set of propositional premises. These premises take the form of assignments of truth values to assertions. The constraints in the TMS are derived from basic axioms of propositional logic. Each of the logical symbols **or**, **and**, **->**, and **not** has an associated axiom set which can be used to generate clausal constraints. The axioms for each of these symbols are given in table 1. For each assertion in the data base which involves one of the basic logical symbols the axioms for that symbol are used to generate the clausal constraints relevant to that assertion. In reading these axioms it is important to remember that each clause gives a means by which each term in that clause might be deduced.

To get some feel for the deductive power of a propositional constraint propagator some examples need be developed. For convenience the scenarios use an **assert** function which takes an assertion

Scenario 1. A Deduction Involving or

```
(assert '(not r))
(NOT R)
(assert '(or r s))
(OR R S)
(truth 's)
TRUE
```

represented as an s-expression and gives the corresponding TMS node a truth value of "true" as a premise. In the scenarios things typed by the user will appear in lower case while responses by the system will appear in upper case.

To understand scenario 1 consider the constraints which are created when the assertions **(or r s)** and **(not r)** are created. Among the constraints generated for **(not r)** is the clause **(v ((not r) . false) (r . false))**. When **(not r)** is made true **r** is deduced to be false via this constraint. In general the axioms for **not** guarantee that an assertion and its negation always have opposite truth values. Among the constraints generated for **(or r s)** is the clause **(v ((or r s) . false) (r . true) (s . true))**. When **(or r s)** is made true this clause is used to deduce that s must be true.

In general there are three assertions which are relevant to an application of one of the logical symbols **and**, **or**, and **->**. The first assertion is the application itself, such as **(or r s)**. The second two assertions are the arguments in this assertion, such as **r** and **s**. The propositional axioms guarantee that whenever a truth value of one of these assertions can be deduced from truth values assigned to the other two, that deduction is made by the TMS. Scenario 2 is another example of the type of deduction which is carried out by the TMS.

2.2. Justifications

Whenever a deduction is made a justification for the deduced value is constructed. Every deduction made by the TMS involves only a single clause, along with the truth values of assertions in that clause. Thus the clause involved in a deduction carries all the information needed for a justification. Therefore in the TMS a justification for a deduced truth value is simply a pointer to the clause which was used to deduce that value.

Every deduced truth value can be associated with a set of supporting truth values via its justification. If any of these values were deduced by the system (i.e. they are not premises) then they will in turn have supporting values. By searching down such support structures it is always possible to find the set of premises from which any deduced value was derived. A query function, **why**, has been defined to give the set of supporting values for any deduced truth value. Scenario 3 gives an example of the use of **why** and figure 1 diagrams the support structure involved. The function **why** can take a numeric argument which refers to the

---

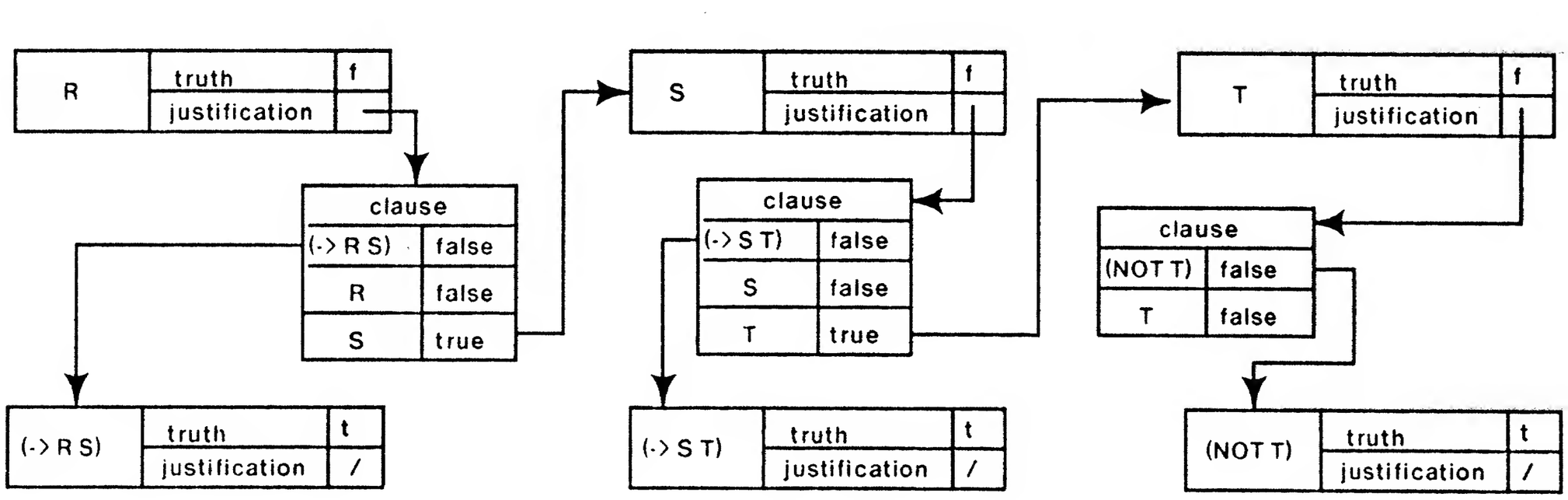
Scenario 2. A Deduction Involving ->

```
(assert '(-> r s))
(-> R S)

(assert '(-> s t))
(-> S T)

(assert '(not t))
(NOT T)

(truth 'r)
FALSE
```

Scenario 3. An example of the use of why

```
(assert '(-> r s))
(-> R S)

(assert '(-> s t))
(-> S T)

(assert '(not t))
(NOT T)

(why 'r)
((R IS FALSE FROM)
 (1 (-> R S) IS TRUE)
 (2 S IS FALSE))

(why 2)
((S IS FALSE FROM)
 (1 (-> S T) IS TRUE)
 (2 T IS FALSE))

(why 2)
((T IS FALSE FROM)
 (1 (NOT T) IS TRUE))

(WHY 1)
((NOT T) IS TRUE AS A PREMISE)
```

Fig. 1. The Support Structure for Scenario 3



assertion which was associated with that number in the last explanation given.

Premises are distinguished in the system as truth values with no justification. At any time the user of the TMS may add any truth value as a premise other than the opposite of a truth value already present *as a premise*. If the added premise is a truth value which was already deduced by the system then the justification for the deduced value is simply removed. If the added truth value is the opposite of one already deduced by the system then the deduced value is retracted and the opposite value is assigned. The details of how this is done and how the resulting contradiction is handled are discussed in the next two sections.

It is important that support structures be "well founded". That is to say that no deduced truth value can depend on itself. For the deduction process as defined so far this is guaranteed since justifications are determined at the instant at which a value is deduced and no value in the system yet depends on the deduced value. However some care need be taken during incremental retraction to see that supports remain well founded. This will be discussed in the next section.

2.3. Retraction

One of the fundamental operations of truth maintenance is incrementally updating the assertional data base when premises are retracted. This should be done in such a way that all deductions are made which would have been made if the system had started with the new premise set, and that every deduced truth value (every one which is not a premise) has a well founded support structure. There are two stages in the retraction process. First all deduced truth values which depended on the removed premise are removed. This is done by checking all clauses which contain any assertion whose truth value was retracted to see if it now invalidly supports some other value. If it does then that value is recursively retracted. The second phase of the retraction process involves checking all assertions which had truth values retracted to see if some value can be deduced for them in some other way. Because all deduction is done in an environment in which all justifications are valid and well founded, the justifications resulting from any deduction must be valid and well founded.

Retraction is also involved when the user adds a premise which the system has already deduced to be false; i.e. the user wishes to assign a truth value to an assertion as a premise but the system has already deduced the opposite value for that assertion. In this case phase one of the retraction process is applied to the deduced truth value so that the deduced value is removed along with all of its consequences. Then the truth value being given that assertion as a premise is added. Finally phase two of the retraction process is applied to perform any deduction which can done via standard propositional constraint propagation. Of course the data base will then be in a state of contradiction, which is discussed in the next section.

2.4. Backtracking and Refutation

The term "contradiction" will be used here to refer to a clause in the TMS all of whose terms are false. For example the clause **(v (p . false) (q . true))** would be a contradiction in any situation in which **p** was true and **q** was false. Contradictions can come into existence at any time during deduction or the addition of premises. The premise set underlying a contradiction is the union of the premise sets for the truth values directly involved. To ensure that backtracking does not interfere with deduction or retraction, all processing of contradictions is done outside any of these processes. When contradictions arise the system asks the user (or the system using the TMS) to choose one of the premises underlying the contradiction for retraction. This process has been termed "dependency directed backtracking" [Stallman & Sussman 77] and a simple example is given in scenario 4.

When a premise which underlies a contradiction is retracted it is important that its negation be deduced to prevent a re-occurrence of the same contradiction at a later time. This can sometimes be done by the propositional constraint propagation algorithm already discussed. However there are cases where this is not so. For example consider what happens in scenario 5. The assertion **c** logically follows from the assertions **(-> a c)**, **(-> b c)**, and **(or a b)**, but the system is incapable of deducing this. When **(not c)** is asserted the system deduces that both **a** and **b** must be false which leads to the clause **(v ((or a b) . false) (a . true) (b . true))** becoming a contradiction (a clause involving one of the the implications could just as easily become a contradiction if deduction was done in a different order). The assertion **(not c)** is one of the premises underlying this contradiction. Without some additional mechanism however the system is incapable of

---

Scenario 4. A Simple Example of Backtracking

```
(assert '(or r s))
(OR R S)

(assert '(not r))
(NOT R)

(assert '(not s))
((THERE IS A CONTRADICTION FROM
  ((OR R S) BEING TRUE)
  (R BEING FALSE)
  (S BEING FALSE))
 (THE UNDERLYING PREMISES ARE
  (1 (OR R S) IS TRUE)
  (2 (NOT R) IS TRUE)
  (3 (NOT S) IS TRUE))
 (WHICH PREMISE SHOULD BE RETRACTED >)) 3

(why 's)
(S IS TRUE FROM
 (1 (OR R S) IS TRUE)
 (2 R IS false))
```

deducing that **(not c)** must be false when the truth of **(not c)** is retracted.

One simple solution is to add a clause which contains the negations of all the premises underlying the contradiction. In the case of scenario 5 this clause is:

(∨ ((or a b) . false) ((-> a c) . false) ((-> b c) . false) ((not c) . false))

Any such generated clause is guaranteed to be a logical tautology because all of the involved premises lead to a contradiction given the clauses already in the system, which are themselves all logical tautologies. Once such a clause is added it can be used to deduce the negation of any one of the premises whenever all the others are believed. Thus when a single premise is retracted its negation is guaranteed to be deducible by the system.

Actually the TMS uses a more complex algorithm which performs local clause resolution. This

---

Scenario 5. Another example of backtracking

```
(assert '(-> a c))
(-> A C)

(assert '(-> b c))
(-> B C)

(assert '(or a b))
(OR A B)

(why 'c)
(I DONT KNOW WHETHER C IS TRUE OR FALSE)

(assert '(not c))
((THERE IS A CONTRADICTION FROM
  (A IS FALSE)
  (B IS FALSE)
  ((OR A B) IS TRUE))
 (THE UNDERLYING PREMISES ARE
  (1 (-> A C) IS TRUE)
  (2 (-> B C) IS TRUE)
  (3 (OR A B) IS TRUE)
  (4 (NOT C) IS TRUE))
 (WHICH PREMISE SHOULD BE RETRACTED >)) 4

(why 'c)
(C IS TRUE FROM
 (1 (NOT C) IS FALSE))

(why 1)
((NOT C) IS FALSE FROM
 (1 (-> A C) IS TRUE)
 (2 (-> B C) IS TRUE)
 (3 (OR A B) IS TRUE))
```

Scenario 6. An Example of Refutation

```
(assert '(-> a b))
(-> A B)

(assert '(-> b c))
(-> B C)

(why '(-> a c))
(I DONT KNOW WHETHER (-> A C) IS TRUE OR FALSE)

(try-to-show '(-> a c))
T

(why '(-> a c))
((-> A C) IS TRUE FROM
(1 (-> A B) IS TRUE)
(2 (-> B C) IS TRUE))
```

---

produces more local constraints which are deductively more powerful and give shorter, more structured justifications when they are used in deductions. However the details of this more sophisticated procedure are irrelevant to the current discussion and the interested reader is referred to appendix one for the Lisp code which performs backtracking in the implemented TMS.

The backtracking mechanism can be used in a refutation technique which increases the deductive power of the system. In refutation the system attempts to deduce a specific truth value for an assertion by first adding the negation of that value as a premise. If no contradiction arises then the attempted deduction fails and the added truth value is removed. If a contradiction does arise then the added value must underlie it and the system deduces the negation of this value in the standard backtracking manner. Thus the desired truth value gets deduced. Scenario 6 gives an example of the use of a **try-to-show** function which invokes the refutation mechanism.

3. PREMISE CONTROL

The TMS can be thought of as an instrument which allows one to view the consequences of a premise set. Being similar to other instruments of examination, the TMS is useful not only in examining given premise sets, but also in determining those premises which are of interest. This Chapter investigates some ways in which the TMS can be used to feed back information from the assertional data base in manipulating the premise set under examination.

A premise controller can be used to automate some manipulations of the premise set. This premise controller can have data structures which are completely independent of the assertional data base. For example the premise controller might associate likelihoods with each potential premise. The user could then manipulate these likelihoods leaving the actual premise control up to the controller. The first two sections of this chapter investigate some methods for doing this type of automatic premise control.

The premise controller can also make use of the assertional data base when choosing premises. For example suppose that the TMS has deduced that the assertion **(taller Bill John)** is true, which is interpreted as saying that Bill is taller than John. In this case it might be useful to assume that Bill is *heavier* than John. The last section of this chapter describes a technique for having this kind of premise control done automatically.

3.1. Assumptions

A simple premise controller can be constructed by making a distinction between solid facts and assumptions. In this system the user would specify a set of solid facts and a set of assumptions. The premise controller would then put both in the premise set. If contradictions arise then the premise controller will always retract an assumption before retracting a solid fact. If there is more than one assumption underlying a contradiction then the user is asked to choose one for retraction. Similarly if a contradiction arises which has no underlying assumptions then the user must choose some "solid fact" for retraction.

An example of the use of assumptions in premise control is given in Scenario 7. In this scenario, and all those that follow, the user is interacting with a premise controller which in turn deals directly with the TMS. Thus the **assert** function tells the premise controller that the given assertion is a solid fact, and the **assume** function tells it that the given assertion is an assumption.

In scenario 7 the premise controller is given two assumptions. First **r** is assumed, which leads to a deduction that **t** is false. Then **t** is assumed and the premise controller makes **t** a premise. This causes the clause which justified **t** being false to become a contradiction. When this contradiction occurs only the assumptions underlying the contradiction are presented to the user, thus the two implications, **(-> r s)** and **(-> s (not t))**, are not considered for retraction. When the assumption **r** is retracted its negation is automatically deduced.

Another example of premise control via assumptions is given in scenario 8. In this scenario three assumptions are made. Logical constraints are placed on these assumptions such that any one of them can be true, but any two of them lead to a contradiction. When **r** and **s** are both assumed a contradiction arises and

Scenario 7. An Example of Premise Control with Assumptions

```
(assert '(-> r s))
(-> R S)

(assert '(-> s (not t)))
(-> S (NOT T))

(assume 'r)
R

(why 't)
(T IS FALSE FROM
 (1 (NOT T) IS TRUE))

(assume 't)
((CONTRADICTION FROM
  (T IS TRUE)
  ((NOT T) IS TRUE))
 (UNDERLYING ASSUMPTIONS ARE
  (1 T IS TRUE)
  (2 R IS TRUE))
 (WHICH ASSUMPTION SHOULD BE RETRACTED)) 2

(why r)
(R IS FALSE FROM
 (1 (-> R S) IS TRUE)
 (2 S IS FALSE))
```

---

the user makes a choice between these two assumptions, leading to a retraction of **r**. When **t** is assumed the user must then choose between **t** and **s**, leading to a retraction of **s**. At this point the user has not expressed any preference between **r** and **t**, both of which were given to the premise controller as assumptions. So the premise controller reinstates **r** as a premise and forces this choice to be made. In general the premise controller never makes an arbitrary choice between assumptions.

3.2. Likelihood Classes

A generalization of the assumption approach to premise control involves placing potential premises in likelihood classes. The assumption approach can be viewed as a special case of this in which there are two likelihood classes, one for known facts and one for assumptions. When contradictions arise in the general likelihood approach, less likely assumptions are always preferred for retraction. The user need only be consulted when there are several premises which tie for being the least likely premises underlying a contradiction. Again the premise controller is very careful not to make arbitrary choices between premises in the same class.

An example of a case in which it might be desirable to have more than two premise classes involves a numerical approximately equal relation, **~**. Such a relation is not truly transitive, i.e. **(~ a b)** and **(~ b c)** does

Scenario 8. A More Complex Example

```
(assert '(not (and r s)))
(NOT (AND R S))

(assert '(not (and s t)))
(NOT (AND S T))

(assert '(not (and  r t)))
(NOT (AND R T))

(assume 'r)
R

(assume 's)
((CONTRADICTION FROM
  (AND R S) IS FALSE)
  (S IS TRUE)
  (R IS TRUE))
 (THE UNDERLYING ASSUMPTIONS ARE
  (1 S IS TRUE)
  (2 R IS TRUE))
 (WHICH SHOULD BE RETRACTED)) 2

(assume 't)
((CONTRADICTION FROM
  ((AND S T) IS FALSE)
  (S IS TRUE)
  (T IS TRUE))
 (THE UNDERLYING ASSUMPTIONS ARE
  (1 S IS TRUE)
  (2 T IS TRUE))
 (WHICH SHOULD BE RETRACTED)) 1

((CONTRADICTION FROM
  ((AND R T) IS FALSE)
  (R IS TRUE)
  (T IS TRUE))
 (THE UNDERLYING ASSUMPTIONS ARE
  (1 R IS TRUE)
  (2 T IS TRUE))
(WHICH SHOULD BE RETRACTED)) 1
```

---

not necessarily imply that **(~ a c)**; otherwise one could prove that things of arbitrarily differing size were roughly equal. However one might want to consider this transitivity to be very likely. One might also have other assumptions in the systems, say about transistor states which are much less certain. Thus one would want at least three likelihood classes used in premise control, one for known facts, one for facts derived from the transitivity of the roughly equal relation, and one for less certain assumptions. The code for a premise controller of this type is presented in appendix two.

3.3. Likelihood Assertions

In addition to retracting premises which lead to a contradiction, a premise controller should be able to use deductions made by the TMS to do more positive types of premise control. For example, if it has been deduced that one person is taller than another, one might want to assume that he is also heavier. Or if one has deduced that some animal is a bird, one might want to assume that it can fly. This type of premise control can be done with likelihood assertions.

Assuming that the a general likelihood class approach is taken to premise control, one can imagine likelihood assertions of the form: **(very-likely p)**, which is intended to mean that the assertion **p** is very likely to be true. There could be a whole range of such likelihood "predicates" such as **somewhat-likely**, **likely**, **very-likely**, etc. Of course it remains to be shown how a system is capable of using (or "understanding") such assertions.

One way of using such assertions is to have the premise controller continually monitor the data base and use them in placing potential premises in likelihood classes. However if the system has assumed that some animal can fly it might be nice to know *why* this assumption was made. While it would be possible to place justification machinery in the premise controller there is a much simpler solution. For each assertion of the form **(likely p)** the system automatically creates the assertion **(-> (likely p) p)**. This implication is then considered to be a "likely" premise by the premise controller. Similar implications would be created for other likelihood predicates and given corresponding status in the premise controller. In this way if the assertion **(likely p)** is ever deduced to be true, the assertion **p** will also become true. The support for **p** will involve the likelihood assertion and it will therefore be clear why **p** is believed. However if the deduction of **p** ever leads to a contradiction then the assumption **(-> (likely p) p)** can be retracted. Scenario 9 gives an example of the use of a likelihood assertion.

Scenario 9. A use of Likelihood Assertions

```
(assert '(-> (bird fred) (likely (flys fred))))
(-> (BIRD FRED) (LIKELY (FLYS FRED)))

(assert '(bird fred))
(BIRD FRED)

(why '(flys fred))
((FLYS FRED) IS TRUE FROM
 (1 (-> (LIKELY (FLYS FRED)) (FLYS FRED)) IS TRUE)
 (2 (LIKELY (FLYS FRED)) IS TRUE))

(why 2)
((LIKELY (FLYS FRED)) IS TRUE FROM
 (1 (-> (BIRD FRED) (LIKELY (FLYS FRED))) IS TRUE)
 (2 (BIRD FRED) IS TRUE))
```

4. INSTANTIATION CONTROL

It is well known that automated deduction and theorem proving systems are subject to explosive computations. However the TMS described in the previous sections seems free of this problem. While it is possible to make the premise controller do a great deal of backtracking (exponential in the number of backtrackable assumptions), in practice this is not important because the number of assumptions is usually small and they do not interact in complex manners. The difference between the TMS and more general deductive systems is that the TMS deals with propositional logic only. All of the difficult problems in automated deduction involve instantiation of quantified formulae and axiom schemas.

From the point of view taken here *instantiation and deduction are separate processes.* Deduction is the process of assigning truth values to assertions based on other truth values already in the system. This can be done entirely by the TMS. Instantiation can be though of as generating propositional formulae upon which the TMS can operate. This outlook on deduction leads to novel control strategies.

4.1. For-All Assertions

Quantified knowledge can be represented in **for-all** assertions. Scenario 10 gives an example of the use of such assertions. In this scenario the assertion **(for-all (dog !x) (mammal !x))** represents the statement that all dogs are mammals. In general a **for-all** assertion has two parts. The first is an "antecedent assertion" such as **(dog !x)** in the example, and is similar to a trigger pattern in a PLANNER demon. Symbols which begin with the character "!" are treated as variables which can match any s-expression. The second part of a **for-all** assertion is a consequent assertion, instances of which follow from the **for-all** assertion and corresponding instances of the antecedent assertion.

For deductions to be made from the assertion **(for-all (dog !x) (mammal !x))** constraints must be placed in the TMS which involve this assertion. TMS constraints for assertions involving the basic logical connectives are derived from axioms for those connectives. Similarly constraints involving **for-all** assertions are derived from implicit axioms for **for-all**. The following constraint is an instance of these axioms. It says that the above **for-all** assertion along with the assertion **(dog Fred)** implies the assertion **(mammal Fred)**.

```
(v ((for-all (dog !x) (mammal !x)) . false)
   ((dog Fred) . false)
   ((mammal Fred) . true))
```

It is clear that it would be impossible to generate all instances of the axioms for **for-all** assertions whenever such an assertion is created in the data base. Therefore some more sophisticated control technique is required for this instantiation process. When a **for-all** assertion is created a demon is constructed which is invoked every time an assertion that matches the antecedent pattern is created in the data base. Each time this demon is invoked it uses the variable binding generated by the triggering assertion to create a corresponding instance of the consequent assertion. Then it adds a constraint, similar to the one above, which states that the **for-all** assertion along with this instance of the antecedent assertion implies the instance of the consequent

Scenario 10. A use of for-all

```
(assert '(for-all (dog !x) (mammal !x)))
(FOR-ALL (DOG !X) (MAMMAL !X))

(assert '(dog fred))
(DOG FRED)

(why '(mammal fred))
((MAMMAL FRED) IS TRUE FROM
 (1 (FOR-ALL (DOG !X) (MAMMAL !X)) IS TRUE)
 (2 (DOG FRED) IS TRUE))
```

---

assertion. It is important to keep in mind that the triggering condition is the creation of an assertion not the assignment of a true truth value to that assertion. Thus the above constraint could have been generated even if the assertion **(dog fred)** was not known to be true. In this case the constraint could be used to deduce that this assertion is in fact false.

4.2. Taxonomic Hierarchies: An Example

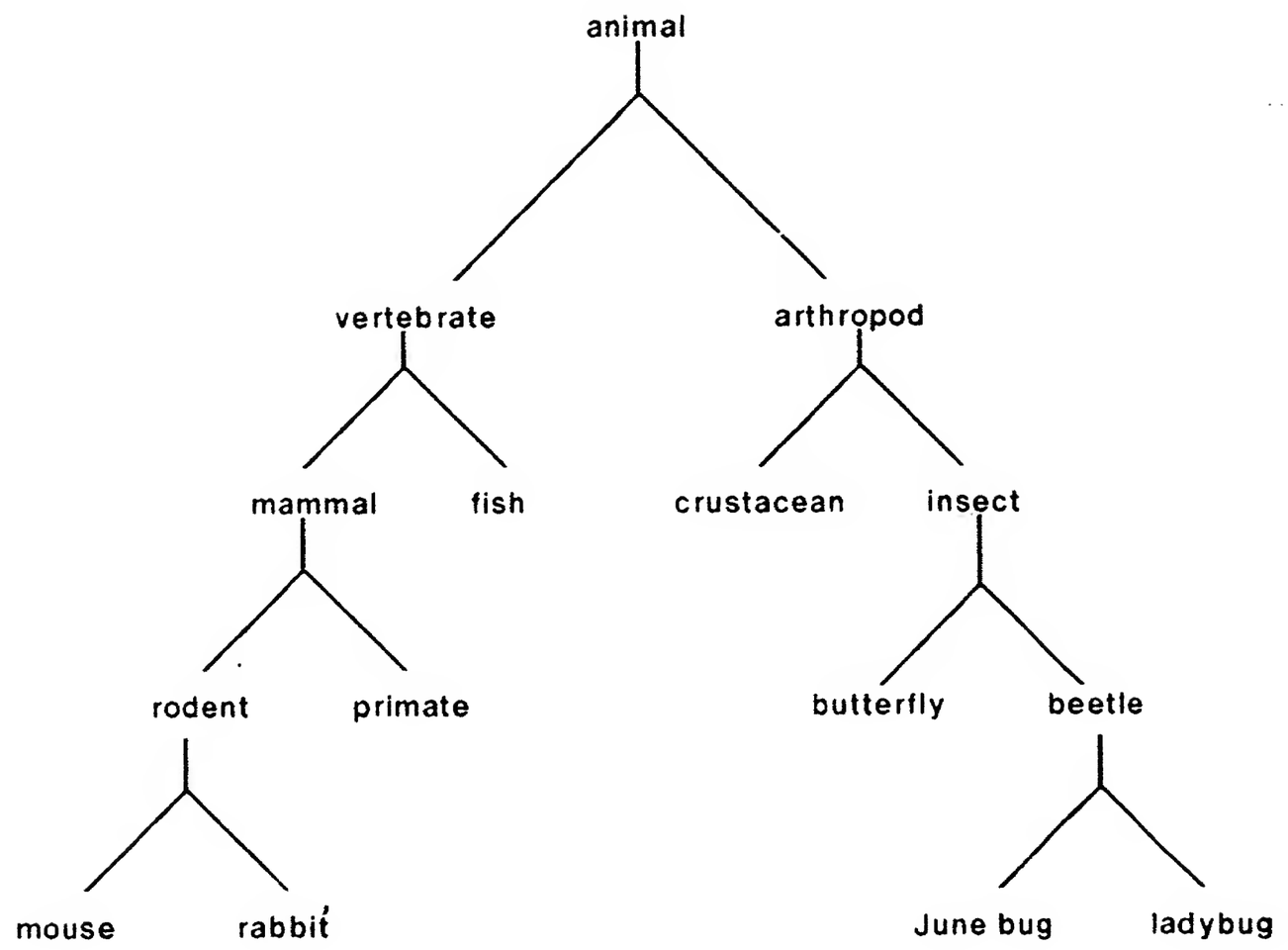
A system for reasoning about taxonomic hierarchies will be developed in this section. Each taxonomic class is represented by a predicate. The taxonomic hierarchy can be represented by assertions of the form:

```
(for-all (cat !x) (mammal !x))

(for-all (dog !x) (mammal !x))

(for-all (dog !x) (not (cat !x)))

(for-all (cat !x) (not (dog !x)))
```

Figure 2 shows a taxonomic hierarchy. A set of assertions similar to the above assertions concerning the predicate **mammal** are asserted for each non-terminal predicate in the hierarchical tree. Scenario 11 gives some sample deductions the system is capable of making from these assertions. When the assertion **(mouse fred)** is created the assertion **(rodent fred)** is also created and a constraint is installed which says that the assertion that all mice are rodents along with the assertion that Fred is a mouse imply the assertion that Fred is a rodent. The creation of the assertion **(rodent fred)** in turn leads to the creation of other constraints and assertions until the top of the hierarchy is reached. Now when **(rodent fred)** becomes true, the TMS automatically deduces that fred is a member of every class superior to **rodent** in the hierarchy.

Now consider what the system does in response to the form **(why '(ladybug fred))**. Since the assertion **(ladybug fred)** did not exist previously in the system the **why** function first creates this assertion. This creation results in the creation of further assertions and logical constraints. These newly created constraints interact with the previous constraints at the level of **vertebrate** and **arthropod**. Since it was asserted in

Fig. 2. A Simple Taxonomic Hierarchy



Scenario 11. Some Deductions Involving the Taxonomic Hierarchy

```
(assert '(mouse fred))
(MOUSE FRED)

(why '(vertebrate fred))
((VERTEBRATE FRED) IS TRUE FROM
 (1 (FOR-ALL (MAMMAL !X) (VERTEBRATE !X)) IS TRUE)
 (2 (MAMMAL FRED) IS TRUE))

(why '(ladybug fred))
((LADYBUG FRED) IS FALSE FROM
 (1 (FOR-ALL (LADYBUG !X) (BEETLE !X)) IS TRUE)
 (2 (BEETLE FRED) IS FALSE))
```

constructing the hierarchy that no vertebrates are arthropods, the system had already deduced that the assertion **(arthropod fred)** was false. This leads to the deduction that Fred is not an insect, and therefore not a beetle, and therefore not a ladybug.

The time it takes it takes to do the types of instantiations and deductions exemplified in scenario 11 is proportional to the depth of the hierarchy which is usually proportional to the log of the number of classes involved. The interesting thing about the above deductions however is not the efficient computation time

involved but instead the simplicity of the control structure used in achieving it. Normally to achieve comparable efficiency one would have to use both forward chaining and backward chaining control structures [Moore 75]. Here a single method for controlling instantiation (as opposed to deduction) fully achieves the desired efficiency.

4.3. Areas for Further Research

The instantiation control techniques which have been discussed so far are only a beginning. Not all domains in deduction lend themselves to so simple a solution as do taxonomic hierarchies. The simple statement that the mother of any person is a person leads to infinite instantiation if a standard **for-all** assertion is used. This section gives some directions for research into other instantiation control techniques. The major point here does not concern specific mechanisms for controlling instantiation, but rather that control of deduction in general should be thought of as control of instantiation.

4.3.1 DEMONIC INVOCATION

In controlling the instantiation of quantified knowledge one would like a convenient way of specifying general conditions under which instantiation is to take place. The axioms for the basic propositional logical operators get instantiated whenever an assertion using one of these operators is created in the data base. This type of demonic invocation will be called "reference invocation". The following is a demon specification utilizing reference invocation which might be used to control the knowledge that the mother of any person is a person:

```
(notice ((assertion-reference (person !x))
         (term-reference (mother !x)))
     (install-constraint '(-> (and (person !x)
                                   (for-all y
                                      (-> (person y)
                                          (person (mother y)))))
                              (person (mother !x)))))
```

Demon specifications of the above form have two parts. First is a set of conditions which must be met for the demon to fire. The second is a body of Lisp code which defines the action taken by the demon when it fires. The trigger conditions of the above demon are both reference conditions which are met whenever a term or assertion matching the given pattern is referenced. When a demon is triggered the variables in the trigger patterns are bound to s-expressions representing terms and assertions. These bindings are then used to replace all occurrences of the variables in the body with the corresponding s-expressions before the body is evaluated. In the above example **install-constraint** is used instead of **assert** because the implication being asserted is really a logical tautology and therefore only a TMS clause equivalent to the implication need be created (no node is generated to represent the implication itself).

Other types of trigger conditions are conceivable. The simplest would trigger whenever assertions of a certain form became true (or false). Assertions and terms might also be assigned "interestingness". Demons

could then trigger when assertions or terms of certain forms are "interesting". Such control states assigned to assertions and terms would be closely related to "control assertions" which have been used in some recent work with pattern directed invocation systems [de Kleer et. al. 77] [Shrobe 79]. However the use of such techniques for instantiation control (as opposed to deduction control) is still largely unexplored.

4.3.2 SPECIAL PURPOSE SUBSYSTEMS

An important aspect of any automated deduction system is the ease with which special purpose subsystems can be incorporated. For example one almost certainly wants some special purpose algorithms for dealing with equalities [Nelson & Oppen 79] [McAllester 80]. In the present context such algorithms can be viewed as controlling the instantiation of the substitution axioms for equality, generating new terms and equating them with other terms in the system. A full discussion of the ways equality can be handled is beyond the scope of this work, but it is important to note that special purpose procedures seem a better option than any attempt to handle it entirely through demonic mechanisms.

Another example of an area in which special purpose subsystems would be desirable is algebraic simplification. There are good algorithms for doing symbolic simplification of algebraic expressions and any reasoning system which must manipulate such expressions should be able to use these algorithms [McAllester 80]. The feasibility of incorporating arbitrary special purpose deduction algorithms and decision procedures is only beginning to be explored.

5. RELATION TO OTHER WORK

The earliest predecessor to the system described here is probably the Davis-Putnam algorithm for determining the satisfiability of sentences in first order predicate calculus [Davis & Putnam 60]. The feature of their algorithm which resembles the TMS is their technique for determining satisfiability of purely propositional sentences. Their method translated the propositional sentence to a set of disjunctive clauses and then performed "propositional constraint propagation" on those clauses much as does the TMS (but they did not call it that). Propositional constraint propagation can also be seen to be very similar to unit clause resolution (resolving first with clauses which contain only a single term). However it is not really appropriate to compare the TMS to any full fledged resolution system since the TMS does not deal with variables (i.e. quantified formula).

These early systems however did not perform the basic TMS functions of justification maintenance, incremental retraction, and dependency directed backtracking. The earliest attempt to handle incremental retraction probably dates back to the "add" and "delete" lists in the STRIPS language [Fikes 71], and PLANNER demons which triggered on the *removal* of assertions [Hewitt 72] [Sussman 71] [McDermott 74]. Later a dependency based mechanism was developed by Richard Fikes for reasoning about state transitions [Fikes 75]. A more sophisticated dependency directed retraction mechanism was later developed by Stallman and Sussman for use in an electrical circuit analysis system [Stallman & Sussman 77]. Their system could make assumptions about transistor states, and when a contradiction was derived, the system uses dependency directed backtracking to track down the particular underlying assumptions.

The first domain independent system which performed all of the basic truth maintenance functions was developed by Jon Doyle [Doyle 78]. Doyle's system used "non-monotonic" dependencies which justify a node being "in" by the fact that some other node is "out". Such dependencies are typically used to make assumptions. For example one might assume A by justifying A with the fact that **(not A)** is out. Thus if **(not A)** ever becomes in, the justification for A will no longer be valid and A will become out. This leads to problems however if the system is able to prove **(not A)** from the assumption of A. First **(not A)** comes in forcing A out. But because **(not A)** depends on A this in turn causes **(not A)** to become out, which, via the non-monotonic dependency, leads to A becoming in, which leads to **(not A)** becoming in, ad infinitum. While there may be ways to fix this problem, it seems hard to motivate the introduction of non-monotonic mechanisms which lead to unnecessary complications.

Another problem with non-monotonic systems is their obscure semantics. Attempts to formalize "non-monotonic logics" are plagued by "unsatisfiable" situations similar to the infinite computation described above [McDermott 78]. While it may be possible to debug these problems, the fundamental motivation behind non-monotonic justifications is suspect. Certainly one cannot argue that an assumption is made *because* one cannot prove its negation. At any time there is an infinite number of assertions which the system can not prove to be false, but one would certainly not want to assume all these things. Therefore a non-monotonic justification does not capture the true reason for making an assumption. It might capture what the system should do if it could prove the negation of an assumption, but this is a backtracking issue and

should not be represented as a justification.

In the truth maintenance system described in this paper as much as possible is done in a traditional framework. The problem with non-monotonic logics is that they bring in non-traditional formalisms too early, muddying deduction, justifications, and backtracking. The aspect of truth maintenance which cannot be formalized in a traditional framework is premise control, which has only just begun to be explored. foofoo

6. APPENDIX I: THE TMS CODE

6.1. The User Interface

These functions interact with the the premise controller. The premise controller works on a priority class scheme which can be initialized to have any number of priority classes, which are assigned consecutive integers from the least to the most certain. The user level functions given here work with three priority classes, numbered 1 through 3. The functions **assert** and **assume** put assertions in the most certain and least certain classes respectively. The middle class is accessed via **very-likely** assertions, which are documented below.

```
(prmcon-init 1 3)

(defun assert (assertion)
   (set-default (referenced-node assertion) 'true 3)
   assertion)

(defun assume (assertion)
   (set-default (referenced-node assertion) 'true 1)
   assertion)

(defun retract (assertion)
   (remove-default (referenced-node assertion)))
```

The assertions are placed in a hash table which is used to insure that no two TMS nodes have the same assertion.

```
(declare (special *assertion-table*))

(setq *assertion-table* (make-array nil 'art-q 4000))

(defun index (form)
  (remainder (hash form) 4000))

(defun referenced-node (assertion)
   (let ((ass (virt-assoc assertion
                          (ar-1 *assertion-table* (index assertion)))))
     (if (cdr ass)
         (cdr ass)
         (let ((node (make-tms-node)))
           (setf (cdr ass) node)
           (setf (assertion node) assertion)
           (instantiate node assertion)
           node))))

(defmacro in-funs (symbol)
   '(get ,symbol 'in-funs))

(defun instantiate (node assertion)
   (if (and (listp assertion) (symbolp (car assertion)))
       (mapc 'funcall (in-funs (car assertion))
                      (circular-list node)
                      (circular-list assertion))))
```

These functions instantiate the basic axioms of propositional logic in the TMS.

```
(defun ->instance (node assertion)
   (let ((n1 (referenced-node (cadr assertion)))
         (n2 (referenced-node (caddr assertion))))
     (add-clause (list (cons node 'false)
                       (cons n1 'false)
                       (cons n2 'true)))
     (add-clause (list (cons node 'true) (cons n1 'true)))
     (add-clause (list (cons node 'true) (cons n2 'false)))))

(addf '->instance (in-funs '->))

(defun or-instance (node assertion)
   (let ((n1 (referenced-node (cadr assertion)))
         (n2 (referenced-node (caddr assertion))))
     (add-clause (list (cons node 'false)
                       (cons n1 'true)
                       (cons n2 'true)))
     (add-clause (list (cons node 'true) (cons n1 'false)))
     (add-clause (list (cons node 'true) (cons n2 'false)))))

(addf 'or-instance (in-funs 'or))

(defun and-instance (node assertion)
   (let ((n1 (referenced-node (cadr assertion)))
         (n2 (referenced-node (caddr assertion))))
     (add-clause (list (cons node 'true)
                       (cons n1 'false)
                       (cons n2 'false)))
     (add-clause (list (cons node 'false) (cons n1 'true)))
     (add-clause (list (cons node 'false) (cons n2 'true)))))

(addf 'and-instance (in-funs 'and))

(defun not-instance (node assertion)
   (let ((n1 (referenced-node (cadr assertion))))
     (add-clause (list (cons node 'true) (cons n1 'true)))
     (add-clause (list (cons node 'false) (cons n1 'false)))))

(addf 'not-instance (in-funs 'not))
```

These functions interface likelihood assertions with the premise controller.

```
(defun likely-instance (node assertion)
   (assume '(-> ,assertion ,(cadr assertion))))

(addf 'likely-instance (in-funs 'likely))

(defun very-likely-instance (node assertion)
   (let ((n1 (referenced-node '(-> ,assertion ,(cadr assertion)))))
     (set-default n1 'true 2)))

(addf 'very-likely-instance (in-funs 'very-likely))
```

```
(defun try-to-show (assertion)
   (let ((node (referenced-node assertion)))
     (refute (cons node 'false))
     (eq (truth node) 'true)))

(defun why (item)
    (if (and (numberp item) (= item 0))
        (pop-query)
        (let ((node (if (numberp item)
                        (answer item)
                        (referenced-node item))))
          (cond ((unknown? node)
                 '(I dont know whether or not ,(assertion node) is true))
                ((null (support node))
                 '(,(assertion node) is ,(truth node) as
                   ,@(cdr (assoc (certainty node) '((1 a likely)
                                                    (2 a very-likely)
                                                    (3 an asserted))))
                   premise))
                (t (push-query (cons '(,(assertion node) is ,(truth node) from)
                                     (fmapcar '(lambda (term)
                                                 (if (not (eq (car term) node))
                                                     (cons '(,(assertion (car term))
                                                             is
                                                             ,(truth (car term)))
                                                           (car term))))
                                              (clause-list (support node))))))))))
```

6.2. The TMS

```
(declare (special *contra-list* *removed-list* *noticers*
                  *premise-selector* *premise-checker*))

(defun tms-init (prem-selector prem-checker)
  (setq *premise-selector* prem-selector)
  (setq *premise-checker*  prem-checker)
  (setq *contra-list* nil)
  '(tms-ready))

(defstruct (tms-node)
  assertion
  (truth 'unknown)
  support
  true-noticers
  false-noticers
  unknown-noticers
  neg-clauses
  pos-clauses
  external-properties)

(defmacro opposite (value)
         '(if (eq ,value 'true) 'false 'true))

(defmacro clauses (node value)
         '(if (eq ,value 'true)
              (pos-clauses ,node)
              (neg-clauses ,node)))

(defmacro op-clauses (node value)
         '(if (eq ,value 'true)
              (neg-clauses ,node)
              (pos-clauses ,node)))

(defmacro noticers (node value)
  '(cond ((eq ,value 'true)
          (true-noticers ,node))
         ((eq ,value 'false)
          (false-noticers ,node))
         (t (unknown-noticers ,node))))

(defmacro unknown? (node)
  '(eq (truth ,node) 'unknown))

(defmac premise? (node)
  (and (not (unknown? node)) (null (support node))))

(defmac true-term? (term)
  (eq (truth (car term)) (cdr term)))

(defmac false-term? (term)
  (eq (truth (car term)) (opposite (cdr term))))

(defmacro unknown-term? (term)
  '(unknown? (car ,term)))

(defmacro op-term (term)
  '(cons (car ,term) (opposite (cdr ,term))))
```

```
(defmacro make-clause ()
  '(cons nil nil))

(defmacro clause-list (clause)
  '(car ,clause))

(defmacro psat (clause)
  '(cdr ,clause))

(defun add-clause (clist)
   (let ((clause (add-2 clist))
         (*noticers* nil))
     (deduce-check clause)
     (run-noticers)))

(defun add-2 (c-list)
  (let ((clause (make-clause)))
       (setf (clause-list clause) (merge c-list nil))
       (mapc '(lambda (term)
                (addf clause (clauses (car term) (cdr term))))
             (clause-list clause))
       (setf (psat clause) (comp-psat (clause-list clause)))
       clause))

(deftail comp-psat (clist)
  (if (null clist)
      0
      (if (not (false-term? (car clist)))
          (1+ (comp-psat (cdr clist)))
          (comp-psat (cdr clist)))))
```

```
(defun make-premise (node value)
   (let ((*noticers* nil))
     (cond ((unknown? node)
            (set-truth node value))
           ((eq value (truth node))
            (setf (support node) nil))
           (t (let ((*removed-list* nil))
                (remove-truth node)
                (set-truth node value)
                (removed-check))))
       (run-noticers)))

(defun set-truth (node value)
  (set-2 node value)
  (mapc '(lambda (noticer)
           (addf noticer *noticers*))
        (noticers node value))
  (mapc 'deduce-check (op-clauses node value)))

(defun set-2 (node value)
  (mapc '(lambda (clause)
           (setf (psat clause) (1- (psat clause))))
        (op-clauses node value))
  (setf (truth node) value))

(defun deduce-check (clause)
  (cond ((= (psat clause) 1)
         (let ((term (unknown-term (clause-list clause))))
           (if term
               (deduce (car term) (cdr term) clause))))
        ((= (psat clause) 0)
         (addf clause *contra-list*))))

(deftail unknown-term (clist)
  (cond ((null clist) nil)
        ((unknown-term? (car clist))
         (car clist))
        (t (unknown-term (cdr clist)))))

(defun deduce (node value sup-clause)
  (setf (support node) sup-clause)
  (set-truth node value))
```

```
(defun retract-premise (node)
   (if (premise? node)
       (let ((*noticers* nil)
             (*removed-list* nil))
         (remove-truth node)
         (removed-check)
         (run-noticers))))

(defun remove-truth (node)
       (let ((value (truth node)))
            (remove-2 node value)
            (addf node *removed-list*)
            (mapc 'retract-check (op-clauses node value))))

(defun remove-2 (node value)
       (if (unknown? node) (break removing-truth-of-unknown-node))
       (mapc '(lambda (clause)
                      (setf (psat clause) (1+ (psat  clause))))
             (op-clauses node value))
       (setf (truth node) 'unknown)
       (setf (support node) nil))

(defun retract-check (clause)
  (if  (> (psat clause) 1)
       (let ((node2 (satisfier (clause-list clause))))
         (if (and node2 (eq clause (support node2)))
             (remove-truth node2)))))

(deftail satisfier (clist)
  (cond ((null clist) nil)
        ((true-term? (car clist))
         (caar clist))
        (t (satisfier (cdr clist)))))
```

All nodes whose support status has changed (the node's previous support was invalidated) are passed to the premise controller which determines if the premises should be changed based on the current support structure.

```
(defun removed-check ()
  (mapc 'node-deduce-check *removed-list*)
  (funcall *premise-checker* *removed-list*)
  (mapc '(lambda (node)
           (cond ((unknown? node)
                  (mapc '(lambda (noticer) (addf noticer *noticers*))
                        (unknown-noticers node)))))
        *removed-list*))

(defun node-deduce-check (node)
  (cond ((unknown? node)
         (node-check-2 node 'true (pos-clauses node))
         (node-check-2 node 'false (neg-clauses node)))))

(deftail node-check-2 (node value clauses)
  (if clauses
      (let ((clause (car clauses)))
        (if (= 1 (psat clause))
            (deduce node value clause)
            (node-check-2 node value (cdr clauses))))))
```

```
(deftail run-noticers ()
    (cond (*contra-list*
           (let ((contra (car *contra-list*)))
                (setq *contra-list* (cdr *contra-list*))
                (if (= 0 (psat contra)) (backtrack contra))
                (run-noticers)))
          (*noticers*
           (let ((next (car *noticers*)))
                (setq *noticers* (cdr *noticers*))
                (eval next)
                (run-noticers)))))

(defun backtrack (contra)
   (let ((prems (premises (clause-list contra))))
     (let ((prem (cond ((null prems) (break contradiction))
                       ((null (cdr prems)) (car prems))
                       (t (funcall *premise-selector* prems)))))
       (let ((path (support-path prem (clause-list contra)))
             (*removed-list* nil))
         (invert path contra)
         (delf prem *removed-list*) ; the premise controller has already selected this node
         (removed-check)))))

(defmacro premises (clist)
  '(merge (premises2 ,clist) nil))

(defun premises2 (clist)
  (if clist
      (if (true-term? (car clist))
          (premises (cdr clist))
          (if (premise? (caar clist))
              (cons (caar clist) (premises (cdr clist)))
              (nconc (premises (clause-list (support (caar clist))))
                     (premises (cdr clist))))))))

(defun refute (term)
   (let (((node . value) term))
     (if (unknown? node)
         (let ((*removed-list* nil)
               (*noticers* nil)
               (*contra-list* nil))
           (set-truth node value)
           (let ((path (support-path node (clause-list (car *contra-list*)))))
             (if path
                 (invert path (car *contra-list*))
                 (remove-truth node))
             (removed-check)
             (run-noticers))
           (if (unknown? node) nil t))
         (print '(warning -- refutation attempted on known truth value)))))
```

The following is a useful utility in choosing premises for retraction

```
(defun user-choice (assums)
  (push-query '((there is a conflict between)
           ,@(mapcar '(lambda (node)
                       (cons '(,(assertion node)
                               assumed to be
                               ,(truth node))
                             node))
                     assums)))
  (print '(which assumption should be retracted?))
  (answer (read)))
```

A support path is a list of nodes such that for any two sequential nodes the latter node is in the support clause for the former. The function **support-path** is used to find a support path from a contradiction to a premise. **(support-path node clist)** returns a support path such that the first node in the path is in **clist** and the path ends with **node**.

```
(deftail support-path (node c-list)
  (cond ((null c-list) nil)
        ((true-term? (car c-list))
         (support-path node (cdr c-list)))
        (t (let ((node2 (caar c-list)))
             (cond ((eq node node2)
                    (list node))
                   ((premise? node2)
                    (support-path node (cdr c-list)))
                   (t (let ((path (support-path node (clause-list (support node2)))))
                        (if path
                            (cons node2 path)
                            (support-path node (cdr c-list))))))))))

(deftail invert (path contra)
  (if path
      (let ((node (car path)))
        (let ((path2 (circular-path node contra)))
          (if path2
              (let ((node2 (car path2)))
                (let ((contra2 (add-2 (resolution (path-resolution path2)
                                                  (clause-list contra)
                                                  node2))))
                  (invert path contra2)))
              (let ((next-contra (support node))
                    (value (truth node)))
                (remove-truth node)
                (deduce node (opposite value) contra)
                (invert (cdr path) next-contra)))))))

(defun circular-path (node contra)
  (support-path node (remove-node node (clause-list contra))))

(defun path-resolution (path)
  (path-resolution2 (cdr path) (clause-list (support (car path)))))

(deftail path-resolution2 (rest-path clist)
  (if (cdr rest-path)
      (path-resolution2 (cdr rest-path)
                        (resolution clist
                                    (clause-list (support (car rest-path)))
                                    (car rest-path)))
      clist))

(defun resolution (clist1 clist2 node)
  (append (remove-node node clist1)
          (remove-node node clist2)))

(deftail remove-node (node clist)
  (if (eq (caar clist) node)
      (cdr clist)
      (cons (car clist) (remove-node node (cdr clist)))))
```

6.3. The Premise Controller

The premise controller is best understood in terms of the invariants it enforces. First a node with a default truth value (one that is in some premise priority class) can have its default value as a deduced value (instead of as a premise) only if all the premises underlying that deduction are in a stronger class. Such a node can take on the opposite of its default value only when the premises underlying that value are in stronger priority classes or when the node has been chosen explicitly by the user for retraction when it conflicts with other premises in its own class.

```
(declare (special *min-cert* *max-cert*))

(defun prmcon-init (minc maxc)
       (tms-init 'prmcon-selector 'prmcon-checker)
       (setq *min-cert* minc)
       (setq *max-cert* maxc))

(defmacro default (node)
     '(cdr (virt-assq 'default (external-properties ,node))))

(defmacro default-certainty (node)
     '(cdr (virt-assq 'default-certainty (external-properties ,node))))

(defun certainty (node)
       (cond ((unknown? node) 0)
             ((premise? node)
              (default-certainty node))
             (t (min-cert (support node)))))

(defmacro min-cert (clause)
   '(min-cert2 *max-cert* (clause-list ,clause)))

(defun min-cert2 (min-cert clist)
    (cond ((null clist) min-cert)
          ((not (false-term? (car clist)))
           (min-cert2 min-cert (cdr clist)))
          (t (min-cert2 (min min-cert (certainty (caar clist)))
                        (cdr clist)))))

(defun set-default (node value certainty)
    (if (not (numberp certainty)) (break (non numeric certainty)))
    (setf (default node) value)
    (setf (default-certainty node) certainty)
    (premise-check node))

(defun remove-default (node)
     (setf (default node) nil)
     (if (premise? node) (retract-premise node)))

(defun prmcon-checker (nodes)
   (mapc 'premise-check nodes))

(defun premise-check (node)
  (if (default node)
      (cond ((or (unknown? node)
                 (not (< (default-certainty node)
                         (certainty node))))
             (make-premise node (default node))))))
```

```
(defun prmcon-selector (premises)
  (let ((assums (least-cert-premises premises)))
    (if (cdr assums)
        (user-choice assums)
        (car assums))))

(defun least-cert-premises (premises)
  (least-cert-2 (list (car premises))
                (default-certainty (car premises))
                (cdr premises)))

(defun least-cert-2 (so-far min-cert rest)
  (if (null rest)
      so-far
      (let ((node (car rest)))
        (let ((cert (default-certainty node)))
          (cond ((< cert min-cert)
                 (least-cert-2 (list node) cert (cdr rest)))
                ((= cert min-cert)
                 (least-cert-2 (cons node so-far) min-cert (cdr rest)))
                (t (least-cert-2 so-far min-cert (cdr rest)))))))))
```

7. APPENDIX II: UTILITY PROCEDURES

Most of the basic concepts behind the utilities described here have been developed by various people other than the author and many of them are documented in the LISP MACHINE MANUAL [Weinreb & Moon 78].

7.1. Basic Macros

7.1.1 BACKQUOTE

The backquote feature provides a form of quote which replaces items preceded by a comma with their value. The following are some examples of the use of backquote:

```
'(foo a ,(+ 1 2))                    evaluates to: (foo a 3)
'(foo ,(list 'a 'b) (list 'a 'b))  evaluates to: (foo (a b) (list 'a 'b))
```

Items in the interior of backquoted expressions which are preceded by ,@ have their values exploded into the top level list structure. An example of the use of this feature is as follows:

```
'(foo ,@(list 'a 'b) ,(list 'a 'b) (list 'a 'b))

        evaluates to:

(foo a b (a b) (list 'a 'b))
```

7.1.2 DEFMACRO

This form is used to define macros. A macro definition has a similar syntax to a function definition. When a form whose car is a macro is evaluated the macro definition is used to generate a new form whose value is the value returned for the original form. The arguments to the macro are bound to the forms in the argument positions rather than their values as is done for functions. An example of a macro definition is given below:

```
(defmacro first-part (x)
      '(caar ,x))
```

Using this definition **(first-part a)** macro expands to: **(caar a)** and so **(fist-part a)** has the same value as: **(caar a)**. A macro is often used instead of a trivial function definition because it is expanded within the compiler and results in more efficient compiled code.

It is sometimes convenient to allow the bound variable list of a macro to be an arbitrary list structure rather than a simple list. In this case atoms in the bound variable list (or bound variable *pattern*, since it need not be a simple list) are bound to corresponding parts of the expression using the macro. For example the new MACLISP form of **do** could have been defined as a macro along the following lines:

```
(defmacro do (variable-bindings (end-test . end-body) . do-body)
          ...)
```

The bound variable list may also be a single atom, in which case that atom is bound to the entire list of "arguments" to the macro.

7.1.3 DEFMAC

defmac is identical to defun with the exception that a macro is created which the compiler can use to open code the function during the compilation of other functions. This is used purely for reasons of efficiency. The open coding is useful in getting the compiler (and other optimization macros such as **deftail)** to perform optimizations which would not otherwise be done. No function defined via **defmac** can be recursive however since this would lead to infinite expansion during open coding.

7.1.4 IF

(if a b c) macro expands to: **(cond (a b) (t c)).**

(if a b) expands to: **(cond (a b)).**

7.1.5 LET

The **let** feature allows structured lambda binding. An example follows:

```
(let ((a 1)
      (b 2))
     (+ a b))

is equivalent to:

((lambda (a b) (+ a b)) 1 2)
```

The **let** macro allows the the bindee of a binding pair to be an arbitrary list structure whose parts are bound to the corresponding parts of the value being bound. This is convenient for dealing with functions which conceptually return more than one value.

7.2. Side Effect Macros

7.2.1 SETF

The **setf** macro gives a general method for side effecting data structures. The following equivalences give some examples of its use:

```
(setf a b)                  is equivalent to:      (setq a b)
(setf (get a b) c)                                 (putprop a c b)
(setf (car a) b)                                   (rplaca a b)
(setf (cdr a) b)                                   (rplacd a b)
(setf (cond (a b) (c d)) e)                        (cond (a (setf b e)) (c (setf d e)))
```

The setf macro macroexpands its first argument. Thus it is possible to use setf in conjunction with macros as is demonstrated below.

```
(defmacro foo (x)
   '(caar ,x))

(setf (foo a) b)        macroexpands to        (rplaca (car a) b)
```

7.2.2 DEFSIDMAC

defsidmac is just like **defmacro** except that it is used to define macros which side effect their last argument and treats that argument position specially. Specifically it defines a macro which will embed the side effect in conditionals as does **setf**. To see how this works consider the following definition of **addf**.

```
(defsidmac addf (x list)
     '(setf ,list (cons ,x ,list)))

(addf x b)              is equivalent to:       (setf b (cons a b))

but

(addf x (if a b c))     is equivalent to:       (if a (addf x b) (addf x c))
```

While it may seem obscure to write code which side effects conditional expressions, the ability to do so can be important when data structure macros expand to conditionals. In such situations it is sometimes convenient to be able to side effect applications of these macros.

7.2.3 INCREMENT

increment is defined by:

```
(defsidmac increment (x)
     '(setf ,x (1+ ,x)))
```

7.2.4 ADDF

addf is defined by:

```
(defsidmac addf (x list)
     '(setf ,list (cons ,x ,list)))
```

7.2.5 DELF

delf is defined as:

```
(defsidmac delf (x list)
   '(setf ,list (delete ,x ,list)))
```

7.2.6 VIRT-ASSOC

This function is like **assoc** except that it is always guaranteed to return a cons whose car is its first argument. Furthermore if there was no such cons originally in the association list then the cons returned is automatically addf'd to the alist. The following is a typical use of **virt-assoc**

```
(defmacro foo (x)
    '(car ,x))

(defmacro other-properties (x)
    '(cdr ,x))

(defmacro bar (x)
    '(cdr (virt-assoc 'bar (other-properties ,x))))

(setf a (cons nil nil))

(setf (bar a) 'bar-val)

;a now is (nil . ((bar . bar-val)))
;(bar a) is now bar-val
```

7.2.7 VIRT-ASSQ

virt-assq is to **assq** as **virt-assoc** is to **assoc.**

7.3. Definition Macros

7.3.1 DEFSTRUCT

The **defstruct** feature is used to define a type of structured object. A **defstruct** definition creates a set of macros. One of these macros is used to create objects of the defined type. The others are used to access the parts of that object. Consider the following example:

```
(defstruct (ship) x-pos y-pos (mass 200))
```

This defines four macros: **make-ship**, **x-pos**, **y-pos**, and **mass**. The **make-ship** macro creates a ship with its mass set to a default value of 200. The following dialogue illustrates a use of these macros:

```
(SETQ HERO (MAKE-SHIP))
{nil nil 200}

(MASS HERO)
200

(SETF (X-POS HERO) 10)
10

(X-POS HERO)
10
```

7.3.2 DEFTAIL

When **deftail** is used instead of defun in a function definition tail recursion optimization is performed on the body of that definition. This feature actually does more than simple tail recursion optimization in that simple accumulations (functions which generate sums, products, or lists recursively) are also converted to iterative forms.

7.3.3 DEFARB

defarb is identical to defun except that it allows the bound variable list to be an arbitrary list expression. The atoms in this expression are bound to the corresponding parts of the list of values to which the defined is applied. The most common use of defarb is to have the bound variable *pattern* be a single atom in which case that atom is bound to the list of arguments to the function. A function so defined can take an arbitrary number of arguments.

7.4. Query Functions

7.4.1 PUSH-QUERY

This function takes a "query", prints a "query list", and pushes information on an internal data structure which is used to "answer" the query. A query is a cons of an "initial query" and a "query-list". The initial query can be any s-expression and is printed as the first part of the printed query. The query list is an association list of s-expressions with arbitrary objects. The printed query consists of the initial query followed by an enumeration of the s-expressions in the query list. The following example should be useful.

```
(push-query (cons '(the items of interest are)
                  (list (cons 'item1 'answer1)
                        (cons 'item2 'answer2)
                        (cons 'item3 'answer3))))

;       which results in the following being printed:

((the items of interest are)
 (1 item1)
 (2 item2)
 (3 item3))
```

7.4.2 ANSWER

This function is only meaningful after a query has been pushed. It takes a single numeric argument and returns the datum that was associated with the corresponding s-expression in the query enumeration. For example assuming the previous query pushed was the above query, the **answer** would yield the following results:

```
(answer 1) => answer1
(answer 2) => answer2
(answer 3) => answer3
```

7.4.3 POP-QUERY

This function pops the query stack such that further calls to **answer** are computed in the context of an earlier query.

7.5. Mapping Functions

All of the standard MACLISP mapping functions have been converted to macros which macroexpand to iterative forms. This allows one to map macros as well as normal functions. These macros also provide a great deal of optimization not normally supplied by the compiler. For example embedded mappings, such as **(mapc 'foo (mapcar 'bar l))**, macro expand into a single iterative form. Some non-standard mapping functions and special forms relating to mapping functions have also been defined.

7.5.1 CIRCULAR-LIST

This function of one argument returns an infinite, self referential list of that argument. This is used to create list arguments to mapping functions. For example a list of symbols could be set to nil with the following expression:

(mapc 'set symbols (circular-list nil))

The mapping macros recognize **circular-list** arguments and produce iterative forms which avoid actually creating the infinite list.

7.5.2 INTEGERS-BETWEEN

This function of two numeric arguments returns a list of all the integers between those arguments inclusive. Thus one could convert an array to a list with the following code:

```
(mapcar 'ar-1 (circular-list array)
              (integers-between 0 (1- (car (dimension array)))))
```

The mapping functions recognize **integers-between** forms and avoid actually creating such a list. Also because nested mappings are merged, the above form could be given as an argument to a second mapping function and the resulting code would be just as efficient as a single iteration over the elements of the array.

The second argument to **integers-between** can be the atom **inf**. This is recognized by the mapping functions which then treat the **integers-between** argument as an infinite list. However **integers-between** actually only creates a finite list when given **inf** as its second argument.

7.5.3 FMAPCAR

This is the same as mapcar except that all null elements are removed from the list returned. Thus:

```
(fmapcar 'foo l1)

is equivalent to:

(mapcan '(lambda (x) (list (foo x))) l1)
```

7.5.4 FORALL

This could have been defined as:

```
(defun forall (list pred)
   (or (null list)
       (and (funcall pred (car list))
            (forall (cdr list) pred))))
```

7.5.5 EXISTS

This could have been defined as

```
(defun exists (list pred)
   (and list
        (or (funcall pred (car list))
            (exists (cdr list) pred))))
```

7.5.6 ACCUM

This could have been defined as:

```
(defun accum (fun list temp-accum)
   (if (null list)
       temp-accum
       (accum fun
              (cdr list)
              (funcall fun (car list) temp-accum))))
```

7.5.7 LSUM, LPROD

These could have been defined as:

```
(defun lsum (list)
   (accum 'sum list 0))

(defun lprod (list)
   (accum 'product list 1))
```

7.6. Miscellaneous Functions

7.6.1 MERGE

This is defined as:

```
(deftail merge (l1 l2)
   (cond ((null l1) l2)
         ((member (car l1) l2) (merge (cdr l1) l2))
         (t (merge (cdr l1) (cons (car l1) l2)))
```

7.6.2 HASH

This is a hashing function on s expressions.

8. FUNCTION INDEX

->instance 22
accum 38
add-2 25
add-clause 25
addf 34
and-instance 22
answer 36
assert 21
assume 21
backquote 32
backtrack 28
certainty 30
circular-list 37
circular-path 29
clause-list 25
clauses 24
comp-psat 25
deduce 26
deduce-check 26
defarb 36
default 30
default-certainty 30
defmac 33
defmacro 32
defsidmac 34
defstruct 35
deftail 36
delf 34
exists 38
false-term? 24
fmapcar 38
forall 38
hash 39
if 33
in-funs 21
increment 34
instantiate 21
integers-between 37
invert 29
least-cert-2 31
least-cert-premises 31
let 33
likely-instance 22
lprod 38
lsum 38
make-clause 25
make-premise 26
merge 39
min-cert 30
min-cert2 30
node-check-2 27
node-deduce-check 27
not-instance 22
noticers 24
op-clauses 24
op-term 24
opposite 24
or-instance 22
path-resolution 29
path-resolution2 29
pop-query 37
premise-check 30
premise? 24
premises 28
premises2 28
prmcon-checker 30
prmcon-init 30
prmcon-selector 31
psat 25
push-query 36
referenced-node 21
refute 28

index 21
remove-default 30
remove-node 29
remove-truth 27
removed-check 27
resolution 29
retract 21
retract-check 27
retract-premise 27
run-noticers 28
satisfier 27
set-2 26
set-default 30
set-truth 26
remove-2 27
setf 33
support-path 29
tms-init 24
true-term? 24
try-to-show 23
unknown-term 26
unknown-term? 24
unknown? 24
user-choice 28
very-likely-instance 22
virt-assoc 35
virt-assq 35
why 23

9. REFERENCES

[Davis & Putnam 60] Martin Davis, Hilary Putnam
"A Computing Procedure for Quantification Theory."
Journal of the Association for Computing Machinery, Vol. 7, pp. 201-215, 1960

[de Kleer et. al. 77] Johan de Kleer, Jon Doyle, Guy Steele, Gerald Sussman.
Explicit Controle of Reasoning.
MIT AI Lab Memo 427 (Cambridge June 1977).

[de Kleer & Sussman 78] Johan de Kleer, Gerald Jay Sussman.
Propogation of Constraints Applied to Circuit Synthesis.
MIT AI Lab Memo 485 (Cambrige, September 1978).

[Doyle 77] Jon Doyle.
Truth Maintenance Systems for Problem Solving.
M.S. thesis (May 1977). Also MIT AI Lab Technical Report 419 (Cambridge, September 1978).

[Doyle 78] Jon Doyle.
A Glimpse of Truth Maintenance.
MIT AI Lab Memo 461a (Cambridge 1978).

[Fikes 75] Richard E. Fikes
"A Deductive Retrieval Mechanism for State Descriptor Models"
SRI AI Technical Note 106

[Fikes 71] Richard E. Fikes, N. J. Nilsson.
"STRIPS: a New Approach to the Application of Theorem Proving to Problem Solving"
Artificial Intelligence 2, 1971, pp. 189-208.

[Hewitt 72] Carl Hewitt
Description and Theoretical Analysis of PLANNER: a Language for Proving Theorems and Manipulating Models in a Robot.
MIT Technical Report 258, 1972.

[London 78] Philip E. London.
Dependency Networks as a Representation for Modelling General Problem Solvers.
Ph.D. thesis U. Maryland, Dept. of Computer Science Technical Report 698 (College Park, Maryland, September 1978).

[McAllester 80] David A. McAllester

The Use of Equality in Deduction and Knowledge Representation

MIT AI Lab Technical Report 520, February 1980.

[McDermott 74] Drew McDermott, Gerlad J. Sussman.

The CONNIVER Reference Manual

MIT AI Lab Memo 259a, 1974.

[McDermott 78] Drew McDermott, Jon Doyle

Non-monotonic Logic I

MIT AI Lab Memo 486, 1978, Also to appear in *Artificial Intelligence 13.*

[Moore 75] Robert C. Moore

Reasoning From Incomplete Knowledge in a Procedural Deduction System.

MIT AI Lab Technical Report 347 (Cambridge December 1975).

[Nelson & Oppen 79] Greg Nelson, Derek C. Oppen

"Simplification by Cooperating Decision Procedures"

ACM Transactions on Programming Languages and Systems, Vol. 1, No. 2, October 1979, Pages 245-257.

[Reiter 79] Raymond Reiter

A Logic for Default Reasoning.

University of British Columbia, Department of Computer Science, Technical Report 79-8.

[Shrobe 79] Howard E. Shrobe

Dependency Directed Reasoning for Complex Program Understanding

MIT AI Lab Technical Report 405 (Cambridge June 1979).

[Stallman & Sussman 77] Richard M. Stallman, Gerald Jay Sussman.

"Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis."

Artificial Intelligence 9 (1977), 135-196.

[Steele & Sussman 78] Guy Lewis Steele Jr., Gerald Jay Sussman.

Constraints.

MIT AI Lab Memo 502 (Cambridge, May 1978). Also Proc. API.79 Conference (Rochester, May 1979).

[Sussman 71] Gerald J. Sussman, Terry Winograd, E. Charniak.

Micro Planner Reference Manual

MIT AI Lab Memo 203a, 1971

[Sussman 77] Gerald J. Sussman.

"Electrical Design: a Problem for Artificial Intelligence Research"

in IJCAI-77, pp. 894-900, 1977.

[Waltz 72] David L. Waltz.

Generating Semantic Descriptions from Drawings of Scenes With Shadows.

MIT AI Technical Report 271, November 1972.

Also in The Psychology of Computer Vision, Patrick H. Winston (ed.), McGraw-Hill, 1975.

[Weinreb & Moon 79] Daniel Weinreb, David Moon.

LISP Machine Manual.

MIT Artificial Intelligince Laboratory, 1979